The StakeCube blockchain : Instantiation, Evaluation & Applications

Antoine Durand, Guillaume Hébert, Khalifa Toumi, Gérard Memmi, Emmanuelle Anceaume

The StakeCube blockchain : Instantiation, Evaluation & Applications

Antoine Durand
IRT-SystemX
Institut Polytechnique de Paris
antoine.durand@irt-systemx.fr

Guillaume Hébert
Atos
guillaume.hebert@atos.net

Khalifa Toumi
IRT-SystemX
khalifa.toumi@irt-systemx.fr

Gérard Memmi
LTCI, Télécom Paris
Institut Polytechnique de Paris
gerard.memmi@telecom-paris.fr

Emmanuelle Anceaume
CNRS/Univ Rennes/IRISA/Inria
emmanuelle.anceaume@irisa.fr

***Abstract*—Blockchains have seen a recent rise in popularity as a generic solution for trustless distributed applications across a wide range of industries. However, blockchain protocols have faced scalability issues in applications involving a growing number of participants. In this paper we instantiate and evaluate StakeCube, a proposal for a scalable shard-based distributed ledger. We further detail and tune a byzantine agreement algorithm suited for StakeCube's sharding structure, and we experimentally study and asses its performance, especially regarding scalability. We were successfully able to run StakeCube with up to 5000 participants, confirming up to 1100 bytes/s of transaction, with a confirmation time starting at 200 seconds. Finally, we use StakeCube in a large scale energy marketplace application, and show that a node running on a Raspberry Pi Zero is able to handle the load without issues.**

***Index Terms*—blockchain, scalability, sharding, Proof-of-Stake, benchmarking, application, smart grid**

I. Introduction

Blockchain technology showed tremendous progress from the early days of the Bitcoin cryptocurrency to providing versatile and useful tools for distributed applications. Arguably, one of Bitcoin biggest achievement is to efficiently bring consensus protocols to the scale of the Internet, through the use of Proof-of-Work (PoW). In that sense, it showed the potential of large-scale distributed trustworthy applications with participants who do not trust each other, and opened a path for the use of such protocols in a wide range of businesses, administrations, or industries [1].

However PoW-based blockchains have inherent downsides, *e.g.* high energy consumption tied to the security level, synchrony assumptions and increased transaction confirmation delay. To deal with those issues, Proof-of-Stake (PoS) blockchains have been proposed, but compared to Bitcoin they have to fulfill the scalability requirement, which proved itself to be quite challenging. Moreover, because they cannot rely on randomness from the PoW, PoS blockchains often have to securely generate themselves their randmoness which creates additional complexity. Recently, a Proof-of-Stake distributed ledger named StakeCube [2] was proposed to tackle the scalability issue, with a solution based on sharding.

StakeCube has a per-block agreement approach, *i.e.* the generation of a new block is started after consensus has been reached on the previous block, and forks are not possible. Roughly speaking, StakeCube is made efficient by spreading all nodes into shards of fixed size which distribute the block generation process. Each shard is splitted into a core set and a spare set, where the core set act as a committee of fixed size $s_{\min}$ for the shard. Shard membership is based on stake ownership, and is made unpredictable and renewed periodically to prevent eclipse attacks.

StakeCube's sharding structure relies on the distributed hash table PeerCube [3], and thus inherits its properties. It is able

to efficiently handle high churn, and randomisation of shards ensure the resistance against Byzantine adversaries.

Because StakeCube is given as a template from a set of ingredients, we first instantiate it using protocols tuned for its setting. Specifically, StakeCube requires a verifiable inter-shard agreement protocol that we derive from the suggested algorithm from Chen & al [4].

Then, to test StakeCube's practical performance and verify its behavior within large-scale applications, we experimentally measure communication cost as well as transaction confirmation time and throughput under varying network size, security parameters and load. To the best of our knowledge, StakeCube is the first blockchain reaching scalability levels well in the thousands of nodes.

Finally, we demonstrate StakeCube's viability with a large scale IoT application: an energy marketplace [5]. In this application energy producers and consumers sell directly to each-others through a blockchain. We implemented this application on StakeCube, and executed it with a large number of nodes, with one Respberry Pi Zero among them. We measured its ressources usage and found it to be largely within capacity.

We made all data related to the experiments available on Github[1].

A. Related Work

Our work is the first one to evaluate StakeCube, however other blockchains with an emphasis on scalability have been proposed. Large scale blockchain experiments includes Algorand [6], Elastico [7] and the Red Belly Blockchain [8]. Runchao Han *et al.* evaluated several blockchains specifically with IoT in mind [9], namely Hyperledger Fabric v0.6 with PBFT, Fabric v1.0 with BFT-SMaRt, Ripple with BFT Ripple consensus, Tendermint with hybrid PBFT and Casper, R3 Corda with BFT-SMaRt. Their work showed that these algorithms do not scale well passed the tens of devices.

There are several works that evaluates Hyperledger Fabric [10]–[12] with up to 32 nodes. Some of these works make use of the Caliper [13] tool, which also supports most of the Hyperledger projects as well as Ethereum.

Blockbench [14] is a framework supporting Ethereum, Parity and Fabrice, although it focuses more on resources utilised by smart-contracts rather than network usage.

II. StakeCube overview

For self-containment reasons, we now recall the main design features of StakeCube.

StakeCube is a proof-of-stake protocol based on the per-block agreement paradigm. It leverages a sharding structure provided by the PeerCube distributed hash table to make the block agreement procedure efficient and robust to adversarial strategies. It uses the UTXO model, meaning that each public key is associated with some amount of stake that can only be spent all at once.

[1]https://github.com/Maschmalow/StakeCube_experiment_data

A. A Sharded Distributed Hash Table (DHT) based on PeerCube

The notion of Sharded DHT is similar to a regular DHT overlay, that is an overlay in which nodes self-organize according to a given graph topology, except that each vertex of the DHT is a set of nodes instead of a single node. More precisely, in PeerCube [3], nodes gather together into shards, and shards self-organize into a DHT graph topology. Shards are built so that the respective common prefix of the identifiers of their members is never a prefix of one-another. This guarantees that each shard has a unique common prefix, that in turn serves as a shard's *label*. The shard's label characterizes the position of the shard in the overall hypercubic topology, as in a regular DHT. Shard size is lower and upper bounded by parameters $s_{\min}$ and $s_{\max}$, respectively. Whenever the size of a shard $\mathcal{S}$ falls under $s_{\min}$, $\mathcal{S}$ merges with another shard to give rise to a new shard whose label is a prefix of $\mathcal{S}$ label. Whenever the size of a shard $\mathcal{S}$ reaches $s_{\max}$, $\mathcal{S}$ splits into two shards such that the label of each of these two new shards is prefixed by $\mathcal{S}$ label. Each shard self-organizes into two sets. The core set is made of $s_{\min}$ random members while the remaining members joins the spare set. The core set is responsible for running the Byzantine agreement protocols in order to guarantee that each shard behaves as a single and correct entity despite malicious participants [15]. Members of the spare set merely keep track of shard state. Joining the core set only happens when some existing core member leaves, in which case the new member of the core set is randomly elected among the spare set.

In PeerCube, peers are required to have a random identifier that defines the shard membership and topology. StakeCube provides these identifiers through a credential system. With each UTXO and block height is associated a credential. Using Verifiable Random Functions, a random value is generated along with each block, which seeds the credentials' randomness. Because PeerCube also require a minimal amount of churn, we make each credential valid only for T blocks, after which it is renewed with block $T+1$ randomness. As a result, nodes may be participating in multiple shards at once, under different roles, and this membership periodically evolve in an unpredictable manner.

B. The StakeCube protocol

In StakeCube stake owners may join and leave their shard during the protocol execution. Hence before each block generation phase there is a shard membership update phase. In this phase, each shard individually agrees on the set of leaving and joining members, and they also run an election to update their core set accordingly.

At this point it is useful to recall that any shard may be required to run an agreement protocol, which is possible only if the proportion of malicious nodes in each core set is below $1/3$. Similarly to nodes, we call such shards honest or correct, and malicious or Byzantine otherwise. In [2] it is shown that the corruption probability for a shard has an exponentially decreasing bound in $s_{\min}$, hence $s_{\min}$ is polynomial in the security parameter. However, ensuring that all shards are

always honest would imply a $s_{\min}$ too large to be practical, especially given that one of our goals is to spread the load across the network. To work around this limitation, StakeCube is made to tolerate a fixed number F of malicious shards.

This design feature is at the center of the block generation phase. A new block is still created by a single shard using a vector agreement protocol. But to ensure correctness despite malicious shards, it has to be validated by other shards. This is the task of an inter-shard agreement protocol run by a random committee of $S_{com} = 3F + 1$ shards.

This agreement is based on rotating leaders, where each leader shard attempts to propose the next block. If the proposed block is voted by the other shards it is then broadcast and appended to the blockchain for every node. The knowledge of this new block allows the nodes to update the credentials and shard membership of all other nodes and to start the next iteration.

III. Concrete instantiation

A. Model

First, we recall and define various parameters here:

- κ is the security parameter. All other variables are polynomial in κ.
- $s_{\min}$ is the size of the core set of shard. We also define $f = \lfloor s_{\min}/3 \rfloor$.
- S_{com} is the size of the shard committee. We also define $F = \lfloor S_{com}/3 \rfloor$.
- T is the credential renewal period.
- μ is the adversary relative share of stake.

The synchrony assumptions in StakeCube are determined by the concrete instantiation of its building blocks. We choose to use agreement algorithms with consistent requirements, that is, deterministic in the partially synchronous model. Hence, our instantiation is set in the same model.

The adversary is Stake-bounded weakly adaptive, as in StakeCube. Specifically, a proportion $\mu < 1/3$ of stake may be owned by malicious nodes, and the adversary is able to dynamically corrupt new nodes, but subject to a delay of T blocks. That is, if the adversary decides to corrupt some node when the i-th block is created, then corruption will be effective when the $i+T$-th block will be created. We recall here StakeCube's properties, which are guaranteed with probability $1 - e^{O(\kappa)}$.

- **Safety.** If honest user i accepts a block B_h^i at height h in its copy of the ledger then, for any honest user j that accepts a block at height h in its copy ledger, $B_h^j = B_h^i$.
- **Liveness.** If a honest user submits transaction tx, then eventually tx appears in a block accepted in the copy of all honest users.
- **Efficiency.** The number of communication rounds needed to create a block is constant in the number of participants and polynomial in κ.
- **Scalability.** The communication cost grows linearly with the number of participants, and the coefficients are polynomial in κ.

In summary, the focus of the conducted experiments will be to validate the last two properties.

We assume that the initial participants received each-others public key untampered (*i.e.* they trust the genesis bloc), and similarly for transactions we assume the integrity of the public key transmitted by the recipient.

B. Modifications

Our implementation exhibits a significant amount of differences with StakeCube, due to improvements or simplifications.

a) Core members sampling: In contrast to StakeCube, the common randomness used to seed the election of the shards core members is derived from the block seed instead of running a common coin tossing protocol. This means that computing the shard composition only depends on the blockchain and the peers join and leave operations.

b) Join operation: Our prototype does not support the join operation of StakeCube. This means that we assume that participating nodes can only leave or join the system by emitting transactions that spend or give them stake, respectively. All honest users that own stake are required to participate in the algorithm and cannot be offline.

As a result, the original shard update phase of StakeCube can be completely removed and the computation of the shard membership can be carried out from the knowledge of the last block. Although this simplification is indeed a major change in the algorithm, we argue that it does not significantly affect the focus of this work, because the communication cost of the shard update phase is dominated by the block generation phase evaluated in this work.

c) PeerCube: Because in our implementation the entire PeerCube structure is replicated by all nodes, we made a few design choices. It is seen as a binary tree, where a path in the tree represents a label and a leaf is a credential. Hence, shard information is stored at the path addressed by its label, and peers are stored at the leaves addressed by their credential. Then, given a PeerCube tree populated with peers, the shards assignation is computed such that:

- The path of any credential goes through a single shard node
- No shard may have less that $s_{\min}$ members
- If a node is a shard then one of its direct child cannot be a shard (*i.e.* due to its size)

Note that this assignation is the unique one that maximises the shard count.

The main difference with the original PeerCube algorithm is the absence of the upper bound on the shard size $s_{\max}$. In that sense, we "split" a shard whenever possible instead of .

We also store at each node the number of peers belonging in the subtree. This allows us to efficiently batch update operations: inserting/deleting a peer with its credential is logarithmic ; updating shards after several modifications only takes one iteration of the tree, with depth limited to the shards nodes.

d) Communication: To distribute communication load, inter-shard communication is defined to happen between the core sets, which is then responsible for forwarding the message to the spare set. However, if the whole core is corrupted, then shards can now mount eclipse attacks against to prevent any honest spare member from progressing indefinitely. To prevent this situation, we require that there is at least one honest core member in each shard. Because StakeCube's security parametrisation has two independent parameters, *i.e.* $s_{\min}$ and F, we can always meet this this requirement.

e) Block diffusion: When a new block is created, it has to be efficiently broadcast to all nodes participating to StakeCube, *e.g.* through a gossip protocol. For ease of implementation, we rely on a simple diffusion protocol based on the sharding structure: Whenever a shard member receives a block for the first time, the node forwards it to all its neighbour shards. Because inter-shard communication is done through core-to-core broadcast, corrupt shards makes no issues regarding this diffusion protocol.

C. Inter Shard agreement

In Stakecube, an algorithm from Chen & al [16] is suggested for the inter-shard agreement. However, running an algorithm with shards instead of nodes as participants required some adaptations, as well as some additional optimisation related to our setting. Critically, we conserve the property of "player-replaceability" which let different players vote at each step. This property specifically allows us to have different players from the same shard act as one. More precisely, we will say that :

1) A shard S has sent a message m to the shard D whenever at least $2f+1$ core members of S have sent m to all the core members of D.
2) A peer p has received a message m from the shard S whenever it has received m from all the core members at least $2f+1$ core members of D.
3) A shard D has received a message m from the shard S whenever at least $2f+1$ core members of D have received m from the shard D.

According to these definition, we can deduce that if shard S sent a message m to shard D, then D will eventually receive m from D, and that eventual synchrony assumptions also apply. That is, inter-shard communication share the same properties and synchrony assumptions than inter-node communication.

Note that if a honest shard sends a message to a corrupted shard, its honest members will receive the message nonetheless, thus allowing them to continue receiving blocks until they end up in an honest shard again.

Moreover, the algorithm from Chen & al. is not entirely optimal for our setting. More precisely, it achieves security against a rushing adversary. In our case, the adversary is only weakly adaptive and has to wait T blocks before corrupting a node, *i.e.* more than the duration of the agreement. This means that security against static adversaries is sufficient for our inter-shard agreement. Because of the rushing adversary, the original algorithm cannot rely on a known leader and has determine leadership *after* block proposals are sent. We can define a leader rotation determined from the initial common randomness (*i.e.* the block seed), and all shards will wait for its proposal. Note that if no proposal is received, the timers will eventually timeout and the next period will start.

The pseudo code for the inter-shard agreement is given in Algorithms 1 and 2. This algorithm is presented in an event-driven style, where each "upon" block describes the processing of a specific event. Note that this does not imply any kind of parallelism in the treatment of events. In fact, in our implementation, message processing is exclusively single-threaded. We define $\mathcal{H}$ the set of hashes values, $\mathcal{V} := \mathcal{H} \bigcup \{\perp\}$, $\mathcal{V}$ the set of blocks and $leader(period \in \mathbb{N})$ a function that returns shards identifiers in a round-robin manner. The **send** primitive sends an inter-shard message to all shards, and **gossip** efficiently diffuses an inter-nodes message to all shards core members.

a) Differences with the original algorithm: Our algorithm ensures that a vote will ever be sent only for one value at each period. Because the leader is known in advance, we are already expecting its proposition to soft vote.

Additionally, when the current period is reached after next voting a value, all other honest nodes should also reach the same period with the same value. Hence we already know that the leader is going to propose it and we can soft vote it in advance. Note that in this case in the original algorithm, the soft voted value also ignores the leader proposal.

Because we can soft vote in advance or just wait for the leader proposal, the "voting" step of the original algorithm becomes empty for us and can be removed. In this algorithm, "value" refer to hashes of block. The block data itself is broadcast when its value is proposed, and we allow soft votes to be sent without knowing the block data.

However the block data is required to be known to cert vote or output it, which ensure than only valid blocks can be committed. This is an implementation of the "one step" extension of the original algorithm.

1) Safety Proof (sketch): First, we should note that honest nodes may send a $CertVote$ or $SoftVote$ only once per period, and they may send up to two $NextVote$ per period but with one them being $v \neq \perp$. Furthermore, because a message needs to be sent by $2f+1$ nodes from the same shard S to be sent by S , the previous remark also applies to shards instead of nodes.

Assume two honest nodes n_1 and n_2 output blocks b_1 and b_2 at periods p_1 and p_2, respectively. w.l.o.g., assume $p_1 \leq p_2$.

a) First case, if $p_1 = p_2$: Safety is proven through a classical quorum argument: Among the $2F+1$ $CertVote$ received by n_1 and n_2, $F+1$ of them comes from the same shards. Hence at least one of them is from the same honest shard that sent the same block $b = b_1 = b_2$.

b) Other cases, if $p_1 < p_2$: First, note that because an honest node received $2F+1$ $CertVote$ for $H(b_1)$, there are $F+1$ honest shards that may only $NextVote$ once with value $H(b_1)$. Hence any value $v \neq H(b_1)$ may have at most $2F$

Algorithm 1: Inter-shard agreement

local variables:
$period \in \mathbb{N}$ the current period
$prevNextVote \in \mathcal{V}$ the value that advanced us to the current period
$knownBlocks \subset \mathcal{H} \times \mathcal{V}$ the hashset of blocks received from gossip.
timer an object that expires after some timeout duration.
procedure *start_period()*
  start *timer*;
  if $prevNextVote \neq \perp \wedge prevNextVote \in knownBlocks.KeySet()$ **then**
    if $leader(period) == self$ **then**
      send $(Propose, period, prevNextVote)$;
      gossip $(period, knownBlocks[prevNextVote])$;
    send $(SoftVote, period, prevNextVote)$;
  else if $leader(period) == self$ **then**
    $newBlock \longleftarrow GenBlock()$;
    send $(Propose, period, H(newBlock))$;
    gossip $(period, newBlock)$;
end
upon *receiving* $(Propose, period, v)$ *from* $leader(period)$ **do**
  if $prevNextVote = \perp \vee prevNextVote \notin knownBlocks.KeySet()$ **then**
    send $(SoftVote, period, v)$;
end
upon *receiving* $(SoftVote, period, v)$ *from* $2F+1$ *distinct senders* **do**
  if $v \in knownBlocks.KeySet()$ **then**
    if *timer not expired* **then**
      send $(CertVote, period, v)$;
    else
      send $(NextVote, period, v)$;
end
upon *receiving* $(CertVote, period, v)$ *from* $2F+1$ *distinct senders* **do**
  wait until $v \in knownBlocks.KeySet()$;
  output $knownBlocks[v]$;
end

Algorithm 2: Inter-shard agreement Pt.2

upon *timer expires* **do**
  if $(CertVote, period, v)$ *has been sent* **then**
    send $(NextVote, period, v)$;
  else
    send $(NextVote, period, prevNextVote)$;
end
upon *receiving* $(NextVote, p, v)$ *from* $2F+1$ *distinct senders* **do**
  if $p \geq period$ **then**
    $prevNextVote \longleftarrow v$;
    increase *timer* timeout;
    $period \longleftarrow p+1$;
    start_period();
end

NextVote and only $H(b_1)$ may have $2F+1$ *NextVote* at period p_1.

Therefore, for all honest nodes at period p_1+1, $prevNextVote = H(b_1)$. For any honest node n, if $prevNextVote \neq \perp$, then n may only $SoftVote$ or *NextVote* their *prevNextVote* value. Hence, only $H(b_1)$ may receive $2F+1$ $SoftVote$, *CertVote* or *NextVote* at period p_1+1.

Finally, we have that only b_1 may be output at period p_1+1, and that for all honest nodes at period p_1+2, $prevNextVote = H(b_1)$. By induction on $p' > p_1$, only b_1 may be output on subsequent periods.

□

2) Liveness Proof (sketch): We have shown that once a value $v \neq \perp$ receive $2F+1$ *NextVote* at period p, it will continue to do so for all period $p' > p$.

For a value $v \neq \perp$ to receive $2F+1$ *NextVote* at period p, it is sufficient that:

1) All honest nodes are at period p
2) $leader(p)$ is honest
3) They all receive the leader proposal and each other's $SoftVote$ before their timer expire.

As a preliminary remark, we can see that for all period $p' \neq 0$ reached by an honest node, all honest nodes will eventually receive $2F+1$ *NextVote* from period $p'-1$. Moreover, for every period p' where *no* block is output, at least one honest node will reach p'.

Assume that no block has been output yet, we can deduce that for the first item to hold, it is sufficient that all honest nodes receive $2F+1$ *NextVote* from period $p-1$ before their timer expires. Hence, assuming that $leader(p)$ is honest, if the honest nodes' timer duration is greater that the time it takes for some messages to be received, the first and third items will hold. Because the second item holds infinitely often, and due to the eventual synchrony assumption, we can deduce that, eventually, all items will holds.

We now have that there is a period p_s and a value v such that for all period $p' \geq p_s$, all honest nodes have $prevNextVote = v$. For a block to be output a period p', it is sufficient that :

1) All honest nodes are at period p'
2) They all receive each other's $SoftVote$ and *CertVote* before their timer expire.

Using the eventual synchrony assumption similarly to the previous paragraph, we can show that these conditions will eventually hold, and that a honest node will output a block.

□

a) Verifiability: If there are $2F+1$ *CertVote* for the a value v at the same period, then only a block b s.t. $H(b) = v$ may be output. Therefore the set of *CertVote* constitute a certificate for the block b which can convince any node of the protocol output.

This can be used to certify a block for an external party knowing the protocol participants, but we also use it as a mean to guarantee totality: Once a node output a block b, broadcasting its certificate will ensure that all honest nodes will eventually output b.

IV. Experiments

The implementation contains approximately 10,000 lines of C++. It runs as a docker image that can be scaled and run on multiple concurrent host machines. Each docker container is allowed only one computing core. We run the experiments across five virtual machines, each with 30Gib of RAM, 6 virtual CPU, and 30 Gib of storage. A trusted setup phase consists for all the nodes in exchanging key material, shared randomness and connection information. Once initialized, the core of the protocol is triggered and all the nodes output their metrics until each node has locally a chain of blocks of a pre-determined size.

A. Parametrization

In Stakecube all nodes store the same instance of the sharding structure provided by PeerCube. This structure represents the shard membership (core and spare) of the whole network. Hence, we can use it to simulate the corruption in StakeCube. To do so, we generate N credentials with μN of them marked as corrupted, compute the shard membership of nodes and then count the number of corrupted shards. For a given μ and f, this gives us a simple approximation of the probability distribution of the number of corrupt shards. That is, we can estimate the appropriate F to ensure correctness for varying μ and f.

Using this method we computed the following security parameters: $f = 5, F = 5, \mu = 0.1$; $f = 5, F = 10, \mu = 0.15$; $f = 7, F = 7, \mu = 0.15$. The full dataset is available in the repository.

B. Limitations

Because the implementation is an early stage prototype, it underperforms in several aspects. Most notably, the networking code had to be kept simple and naive. For instance, each protocol message is sent in a dedicated TCP connexion, which creates a large overhead. We also did not implement a mechanism for nodes to synchronise and update their states when they communicate. For instance, this means that nodes will keep exchanging outdated blocks even when it is not necessary.

In that respect, we expect that a fully optimised implementation to exhibit a large performance gain, and our experimentation can only be a proof of viability. But despite these issues, we were able to run experiments with up to five thousands nodes, and to keep reasonable transaction output even above a thousand nodes.

C. Metrics

For each block, we output the following metrics, aggregated over nodes:

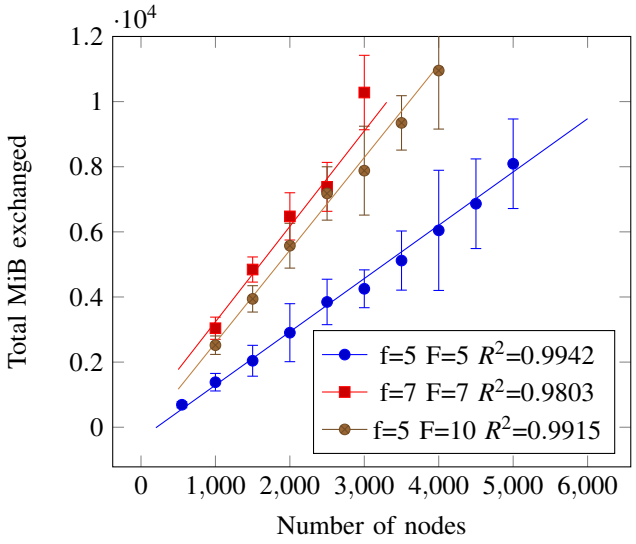

Figure 1. Communication cost as a function of the number of nodes when the number of transactions per block is 10.

1) The median of the block interval time.
2) The sum of the number of bytes sent.

For a given run, the values are averaged over blocks and plotted for varying parameters, namely:

1) Number of nodes. This parameter is set as the 'x' axis to allows us to verify StakeCube's scalability.
2) Block size. The experiments with a small amount of transactions per block indicate StakeCube minimal costs, and the biggest amounts has been chosen to optimise transaction throughput, intended to be representative of operations under maximum load.
3) Security parameters, as chosen in section IV-A.

The error bars represent the standard deviation of the metric for the run. Lines in Figure 1 are linear regressions, with the coefficient of determination R^2 in the legend. Note that because the (maximum) number of transactions in a block is fixed, the transaction throughput can be obtained from the block interval time.

In Figure 1 we show the number of bytes exchanged in the network, *i.e.* the communication cost. We can see that the scalability property of StakeCube clearly holds, as shown by the linear regressions with $R^2 \geq 0.98$. Thus we can conclude that StakeCube is able to tackle the scalability issue of PoS blockchains.

Figure 2 shows the block interval time as a function of the number of nodes in the system. Due to the high variance, we cannot conclude that we achieved a constant number of communication rounds per block. Furthermore, even without variance, the block interval time could increase with the number of nodes solely because the network is slowing down. However the fact that experimental variations have much more influence than the number of nodes is a piece of evidence that our algorithm performs correctly.

We observed during the experiments that the variations were

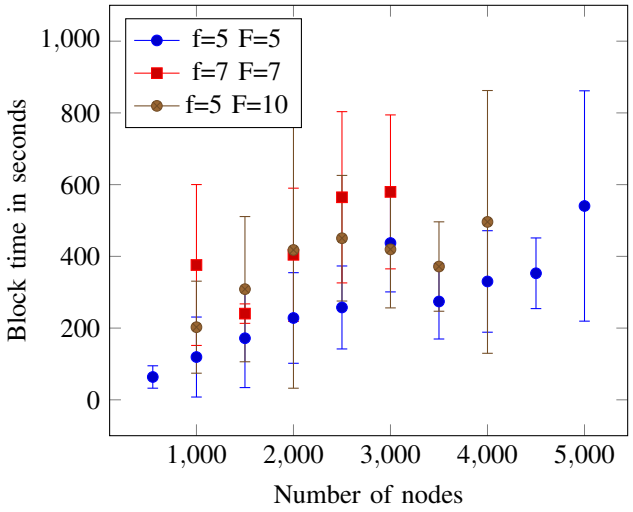

Figure 2. Block time as a function of the number of nodes when the number of transactions per block is 10

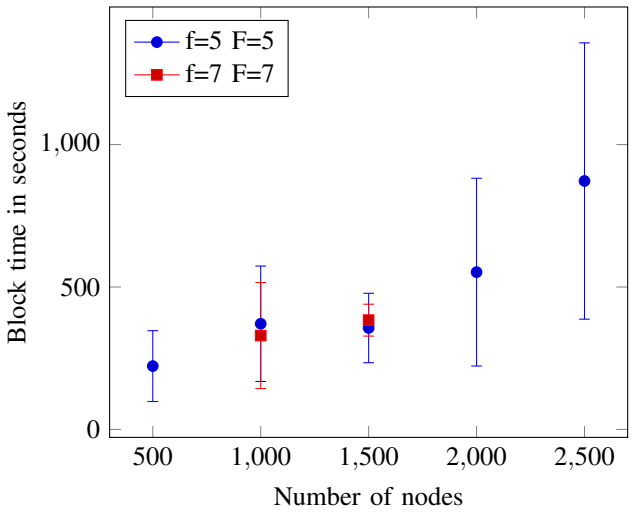

Figure 3. Block time as a function of the number of nodes when the number of transactions per block is 2,000

mostly due to sporadic communication failures between nodes, which happens when the network and the operating system become overloaded. Unfortunately, we were not capable to address the real causes of such experimental variations.

In Figure 3 we show again the block interval time but with blocks containing 2,000 transactions of 128 bytes each. This Figure shows a transaction output between 1,100 and 300 bytes/s. We did not show the communication cost for 2,000 Tx/block because it is very similar to a scaled version of Figure 1 and does not give any additional information.

V. Application

To demonstrate StakeCube's viability in large-scale IoT application, we run an energy marketplace backed by StakeCube on a Raspberry Pi Zero, and confirm that it is able to handle the load even over large networks. The energy marketplace is a blockchain-based answer for the demand of decentralized solutions for the energy infrastructure [5]. In this application, energy producers and consumers such as households and electric vehicles are trading electricity on an local energy marketplace. Because participants may be numerous and are typically running in a small system-on-chip, we see this application as particularly fitting for StakeCube. Indeed, we implemented an energy marketplace pseudo-smartcontract where each node can sell and buy an electricity token and the orders are matched using a double auction algorithm [17].

We ran this application in a network of 500 nodes including one Raspberry Pi Zero, which has 512Mib of RAM and a 1GHz single-core CPU. During execution, we saw that the StakeCube executable used at most 15MiB of RAM and that except for short spikes when receiving blocks the CPU was mostly idling. The full monitoring trace is available in the repository.

VI. Conclusion & Future work

In this paper we presented an instantiation of a sharded distributed blockchain, StakeCube. This work includes a consensus algorithm suited for inter-shard coordination, and computation of security parameters. We then evaluated its performance and its scalability. We finally implemented an energy marketplace application using StakeCube on a Raspberry Pi Zero, which demonstrated its viability for resources-constrained devices within a large network. For future work, beyond improving the implementation, we aim at extending StakeCube's supported features, such as the join system.

Acknowledgements

We are thankful to Louis Martin-Pierrat and David Leporini for their help and fruitful discussions. This work was carried as part of the Blockchain Advanced Research & Technologies (BART) Initiative and the Institute for Technological Research SystemX, and therefore granted with public funds within the scope of the French Program *Investissements d'Avenir*.

References

[1] E. Barka, C. A. Kerrache, H. Benkraouda, K. Shuaib, F. Ahmad, and F. Kurugollu, "Towards a trusted unmanned aerial system using blockchain for the protection of critical infrastructure," *Transactions on Emerging Telecommunications Technologies*, p. e3706, 2019.

[2] A. Durand, E. Anceaume, and R. Ludinard, "Stakecube: Combining sharding and proof-of-stake to build fork-free secure permissionless distributed ledgers," in *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 11704. Springer, 2019, pp. 148–165.

[3] E. Anceaume, R. Ludinard, A. Ravoaja, and F. Brasileiro, "PeerCube: A Hypercube-Based P2P Overlay Robust against Collusion and Churn," in *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2008.

[4] J. Chen, S. Gorbunov, S. Micali, and G. Vlachos, "Algorand agreement: Super Fast and Partition Resilient Byzantine Agreement," Cryptology ePrint Archive, Report 2018/377, Tech. Rep., 2018.

[5] J. Horta, D. Kofman, D. Menga, and A. Silva, "Novel market approach for locally balancing renewable energy production and flexible demand," in *2017 IEEE International Conference on Smart Grid Communications, SmartGridComm 2017, Dresden, Germany, October 23-27, 2017*. IEEE, 2017, pp. 533–539.

[6] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine Agreements for Cryptocurrencies," in *Symposium on Operating Systems Principles (SOSP)*, 2017.

[7] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016, pp. 17–30.

[8] T. Crain, C. Natoli, and V. Gramoli, "Evaluating the red belly blockchain," *CoRR*, vol. abs/1812.11747, 2018. [Online]. Available: http://arxiv.org/abs/1812.11747

[9] R. Han, G. Shapiro, V. Gramoli, and X. Xu, "On the performance of distributed ledgers for internet of things," *Internet of Things*, p. 100087, 2019.

[10] P. Thakkar, S. Nathan, and B. Viswanathan, "Performance benchmarking and optimizing hyperledger fabric blockchain platform," in *MASCOTS 2018, Milwaukee, WI, USA, September 25-28*, 2018, pp. 264–276.

[11] Q. Nasir, I. A. Qasse, M. W. A. Talib, and A. B. Nassif, "Performance analysis of hyperledger fabric platforms," *Security and Communication Networks*, vol. 2018, pp. 3 976 093:1–3 976 093:14, 2018.

[12] H. Sukhwani, N. Wang, K. S. Trivedi, and A. Rindos, "Performance modeling of hyperledger fabric (permissioned blockchain network)," in *17th IEEE International Symposium on Network Computing and Applications, NCA 2018, Cambridge, MA, USA, November 1-3, 2018*. IEEE, 2018, pp. 1–8.

[13] T. L. Fundation, "Hyperledger caliper," accessed April 2020. [Online]. Available: https://www.hyperledger.org/projects/caliper

[14] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K. Tan, "BLOCKBENCH: A framework for analyzing private blockchains," in *Proceedings of the ACM SIGMOD Conference 2017, Chicago, IL, USA, May 14-19*. ACM, 2017, pp. 1085–1100.

[15] E. Anceaume, R. Ludinard, and B. Sericola, "Performance evaluation of large-scale dynamic systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 39, no. 4, 2012.

[16] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 51–68.

[17] K. Brousmichc, A. Anoaica, O. Dib, T. Abdellatif, and G. Deleuze, "Blockchain energy market place evaluation: An agent-based approach," in *IEEE IEMCON*, 2018, pp. 321–327.